Optimizing Higher-Order Functions in Scala

Iulian Dragoş[1]

École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. Scala relies on libraries as the default mechanism for language extension. While this provides an elegant solution for growing the language, the performance penalty incurred by call-by-name parameters, boxing and anonymous functions is high. We show that inlining alone is not enough to remove this overhead, and present a solution based on decompilation of library code combined with inlining, dead code elimination, and copy propagation. We evaluate our approach on two language extensions, Java-like `assert` and C-like for-loops and show improvements of up to 45%.

1 Introduction

Scala[6] is a statically typed, object-oriented language that brings in functional programming elements, like higher-order functions and pattern matching. Scala is designed to be a *scalable* language, a language that can be used both for small programming tasks and for building large systems. The ability to grow the language through libraries is a key aspect. Scala syntax allows users to write code that looks like built-in features of the language, keeping the language small. For instance, the standard library provides a `BigInt` class that is indistinguishable from the standard `Int` type, and the `for` loop on integers is provided through a `Range` class.

This approach is elegant and gives users the power of language designers. However, everything comes at a price, and in this case the price is efficiency. Familiar looking code, like an assert statement or a for loop may conceal unexpected complexities. While library designers are usually aware of these implications, users are often surprised by such performance hits.

Traditional optimizations techniques, like inlining are not very effective. Code is often spread across several compiled libraries, and many short-lived objects on the heap make analysis harder. The Scala compiler is complementing the traditional optimizations with a boxing optimization (that removes unnecessary boxing and unboxing operations), a type-propagation analysis (that obtains more precise types for values on the stack and local variables) and a copy-propagation optimization (that has a simple heap model handling common object patterns like boxed values and closures). All these would be of little use for library code, unless the compiler had a way of analyzing libraries. The Scala compiler is using an *icode reader* for reading back compiled code into the intermediate representation on which the optimizations are run.

The next section shows an example and discusses the sources of inefficiencies. Section 3 discusses the proposed solution and looks into the difficulties we faced in implementing it. Section 4 presents a preliminary evaluation and the last section concludes and presents directions for future research.

2 Example

2.1 Elements of Scala syntax

Before we show the examples, we quickly mention a few language features that will be needed later. For an in-depth description of the language, we recommend [6] or [3]. Scala is a class-based, object-oriented language. It's syntax should be familiar to Java and C++ programmers, although some familiar concepts are expressed slightly differently.

```
class List[A] extends Seq[A] {
  private var len: Int = 10
  def length: Int = len
  def map[B](f: A => B): List[B] = //..
  def foreach(f: A => Unit): Unit = //..
  //..
}
val xs: List[Int] = ..
xs foreach { x: Int => println(x) }
for (x <- xs) println(x)
```

Fig. 1. A Scala syntax show case

Figure 1 shows part of a `List` implementation. `List` takes a type parameter, the type of its elements, and extends `Seq`, the type of sequences of type `A`. Type parameters are introduced by square brackets, instead of angle brackets in Java/C++. Fields or local variables are introduced by **`val`** or **`var`**, the first one denoting a value that cannot be assigned after it has been initialized. Types are introduced by colon, and in many cases can be omitted; the type inferencer fills in the most precise type. Methods are introduced by **`def`** and may omit parenthesis when they take no parameters. Higher-order functions take as parameters or return other functions. Both `foreach` and `map` are higher-order functions[1].

The next lines show two ways in which `foreach` can be called. The first one is a direct call, making use of Scala's support for infix calls: when a method takes a single parameter, the user may omit the dot and the parenthesis around the argument. The argument is a unary anonymous function, delimited by braces.

[1] Although technically they are methods, we use the established terminology in functional programming and call them *higher-order functions*.

The arrow separates the parameter section from the function body. As mentioned before, the type annotations may be dropped. The last line has exactly the same meaning, but uses for-comprehensions to iterate over the elements of `xs`. The compiler desugars it to the previous line.

For-comprehensions are a very general mechanism for operating on sequences. In its most general form a for-comprehension may contain any number of *generators* and *filters*. For instance,

```
for (i <- xs; j <- ys; if (i % j == 0)) print(i, j)
```

prints i and j only when i is a multiple of j. The first two statements are generators, that bind i and j to each element of xs and ys respectively. This is achieved by translating the given comprehension into a series of `foreach`, `filter` (braces were omitted for clarity):

```
xs.foreach(i =>
  ys.filter(j => i % j == 0).foreach(j => print(i, j)))
```

2.2 Examples

We begin by showing how the `assert` keyword from Java can be implemented in Scala as a library function. The tricky part consists in the requirement that the condition and message should not be evaluated unless assertions are enabled, and the assertion fails, respectively. In a strict language like Java, and in the absence of macros, this is impossible to express in the language. Scala offers call-by-name parameters, which turn arguments at each call site into nullary functions. The body of the method applies the function each time it needs to obtain the value.

```
def assert(cond: => Boolean, message: => Any) {
  if (assertionsEnabled && !cond)
    throw new AssertionError("assertion failed: " + message)
}
```

Fig. 2. Predef.assert

Figure 3 shows how call-by-name parameters are compiled. Calling assert implies loading two new classes and instantiating an object for each, which might never be used, or if it is used, it is collect-able immediately after returning from `assert`. What makes things worse, is that this happens at each call site: clearly, such code is going to put a lot of pressure on the garbage collector, and the benefits of not evaluating arguments might be lost completely.

Closures are compiled by turning them into classes that implement one of the `FunctionN` interfaces, where N is the arity. The captured environment is saved as fields of the closure class, and passed as arguments to the constructor. A slight

```
def sqrt(x: Int): Int = {
  assert(x >= 0, "Expected positive integer: " + x)
  //..
}

def sqrt(x: Int): Int = {
  assert(new anonfun1(x), new anonfun2(x))
  //..
}

final class anonfun1 extends Function0[Boolean] {
  private val x: Int = _
  final def apply: Boolean = x >= 0
}
final class anonfun2 extends Function0[String] {
  private val x: Int = _
  final def apply: String = "Expected positive integer: " +  x
}
```

Fig. 3. Using assert

complication is caused by environments that contain **var**s. In order to allow the closure to modify that value, **var**s are wrapped in a reference object that provides methods for changing the value it holds. For simplicity, none of the examples in this paper use mutable variables.

Our second example is looking at for loops on integer ranges, as seen in most imperative languages. As mentioned before, Scala has no special compiler support for such loops, providing a library class that implements `foreach` and `filter`. Therefore, the ubiquitous for loop on integers relies on the for-comprehension machinery, requiring a lot of temporary objects and indirection.

Figure 4 shows an example, including desugaring. In addition to for comprehension desugaring, this example shows another feature that allows users to extend the language in seamless ways: `1 until 10` is translated to a method call on the `Int` class, and since there is no method `until` in `scala.Int`, an implicit conversion is applied first. Method `intWrapper` wraps a primitive integer into a `RichInt`, which has the required methods for creating ranges. The result of `intWrapper(1).until(10)` is a `Range` object that supports iteration. Note that each integer in the loop needs to be unboxed before being applied to the body of the loop.

3 Optimizations

The Scala compiler has several optimization phases that target such cases. The goal is to improve both running time and code size. As shown in the examples above, higher-order functions and anonymous functions are very common,

```
def loop {
  for (i <- 1 until 10) print(i)
}
// desugars to
def loop(): Unit =
  intWrapper(1).until(10).foreach(new anonfun3())

final class anonfun3 extends Function1[Unit, Int] {
  final def apply(i: Int): Unit = Predef.print(Int.box(i));
  final def apply(x1: Object): Object = {
    apply(scala.Int.unbox(x1));
    scala.runtime.BoxedUnit.UNIT
  }
}
```

Fig. 4. For loops on integer ranges.

making it important to keep their numbers down by optimizing them away in statically-known contexts. This can improve running time as well, since class load time and verification on the Java VM are expensive.

The optimization phases are ran on an intermediate representation (ICODE) that is close to the target code (Java bytecode or .NET IL). ICODE is stack based, and uses a control–flow graph of basic blocks to facilitate data flow analysis. One of its goals is to make it easy to read back ICODE from compiled code.

In a first phase, methods are inlined. Then a closure elimination tries to remove all references to the closure object (or environment). A dead–code elimination pass removes unnecessary assignments, mainly coming from the inlining phase, and all unreferenced closure objects (including their class).

3.1 Inlining

The inlining phase uses a fairly expensive analysis to derive precise types for local variables and stack positions. It is an intraprocedural, forward data flow analysis, which uses the Scala type lattice to derive values at each program point. The goal of this analysis is to resolve method calls. Since we are not relying on a whole–program analysis, the inliner can inline only final methods. This is not a problem in our main cases of interest, closures: they are always final classes that extends a `FunctionN` interface.

Class Hierarchy Analysis (CHA) and Rapid Type Analysis[2, 1] are reported in literature to give very good results in practice. It proceeds by removing from the class hierarchy those classes that are never instantiated. However, in our case this does not help, as each anonymous class is instantiated exactly once. Therefore, all method calls to a `FunctionN` apply method are truly polymorphic. Instead, our analysis aims to derive the most precise type possible for the receiver object. It turns out that in most cases this is the anonymous function type, which is final and statically known.

The decision to inline is taken based on method size (calls inside small methods are usually poor choices, as the Java VM native compiler favors small methods). Higher–order methods are preferred for inlining, as they usually allow complete elimination of their argument. In Figure 4, inlining method `foreach` allows, in a next step, to inline the anonymous function given as argument (`anonfun3`).

The analysis and inlining phase are repeated until a fix point is reached (no more methods can be inlined) or a size limit is reached. This ensures reasonable compilation times and method sizes (as mentioned before, large methods take a penalty hit as the JIT compiler seems to be more reluctant to compile them).

3.2 Closure elimination

This phase is trying to infer what values passed in the closure environment can be accessed from the method environment. In other words, the analysis tells whether fields of the anonymous class can be proved to be the same as some local variable available in the enclosing method. This case is common after inlining both a higher order function and the closure's `apply` method.

The analysis is similar to reaching definitions, but it adds a simple model of the heap: in addition to local variables and stack positions, values of objects on the heap that are reachable from locals or stack are modeled as simple records. Such records are populated when known constructors are invoked. For now the only constructors that can create non–empty records on the heap are closure constructors (which populate the record with the captured environment), case class constructors (which populate the record with the given case arguments) and box methods (which populate the record with a single field, the boxed value).

Once the analysis has determined copy–of relations between locals and values on the heap, it proceeds at replacing them by the cheapest operation. At the end of this phase, boxed and closure objects may be unreferenced, and it is the next phase which cleans them up.

3.3 Dead–Code Elimination

The last optimization phase is cleaning up dead code. It uses a standard mark and sweep algorithm: in a first phase, useful instructions are marked, then in a second phase all instructions not marked are removed. Special care must be taken to keep the stack valid, by adding necessary `DROP` instructions. Some of them might prove useless later, and a peephole optimizer is run as the very last operation before code generation.

To mark useful instructions, the algorithm starts with instructions that are known to be needed, like `RETURN` and side–effecting methods. Then, based on reaching definitions, it recursively marks all instructions that create the definitions used by marked instructions. If at the end of this phase there are unreferenced closure classes, they are removed completely.

Test Case	Running time (ms)	Optimized (ms)	Speed up
assert	104.8 (0.84)	67.4 (0.55)	36%
assert (disabled)	79.6 (0.55)	44.4 (0.55)	44%
matrix	75.4 (0.55)	40.4 (0.55)	46%

Fig. 5. Evaluation: running time improvements. Times are average over 5 runs, standard deviation is given in parenthesis.

3.4 Icode Reader

In order to handle library code, the optimization phases have to be able to get back an ICODE representation. The current Scala compiler has a Java bytecode reader that is integrated in the optimization phases[2].

The Java bytecode reader has to solve several aspects in order to integrate with the optimization passes: symbol recovery and typing locals. Symbol recovery means that each symbol coming from the bytecode has to be resolved in the compiler's symbol table. This is not trivial, as compilation of Scala–specifics like traits is fairly involved: each trait generates an interface and an implementation class. Reading back one of them should actually read both, and reuse existing symbols whenever the same entity is referred. For example, reading back a method implementation that calls a method that is currently compiled should reuse the symbol that the compiler created in the earlier phases.

Unlike Java bytecode, ICODE uses typed local variables. When reading code using the same location for values of different types, the bytecode reader splits the local in two different locals. The approach is similar to the approach taken by the Soot framework[7].

4 Preliminary Evaluation

In this section we present preliminary results using the described optimizations. We believe these results are only preliminary, and significant improvements are still possible. However, they are still mini benchmarks, and we don't expect speedups of this scale on usual programs. A more thorough evaluation is necessary to truly assess the effectiveness of these optimizations.

Time measurements are done using `System.currentTimeMillis`, and each benchmark has been run once to warm up the VM before the actual measurement was taken. All tests were run on an Intel Core Duo at 2.5 GHz processor with 2 GB of RAM, running MacOS 10.5.2 and Java 1.5.0_13. Each measurement is the mean over 5 runs, with the standard deviation shown in parenthesis.

[2] In order to keep compilation times low, it is enabled only on certain scala libraries. Once it becomes more mature and performance issues are solved, it will be enabled on all code.

4.1 Assert

The test case for the assert method was to run 500,000 times code containing several simple assertions, with assertions disabled and then enabled. The code is shown in Figure 6. Bytecode inspection showed that closures for both arguments have been eliminated completely.

4.2 For loops

For evaluating for–loops performance we used a simple integer matrix multiplication algorithm. The code is shown in Figure 6 (as Scala compiles to the Java Virtual Machine, it inherits its model of arrays: multi-dimensional arrays are implemented using one-dimensional arrays). We used a slightly modified version of the Scala `Range` class, in order to have `foreach` marked final. This version is likely to be included in the next Scala release.

```
def multiply(n: Int, a: Array[Array[Int]], b: Array[Array[Int]]) {
  val c: Array[Array[Int]] = new Array(n, n)
  for (i <- 1 until n; j <- 1 until n; k <- 1 until n)
    c(i)(j) += a(i)(k) * b(k)(j)
}

// assert test
def test(xml: Node) {
  assert(xml.label == "persons", "Wrong Xml root element: " + xml)
  test2(xml.label)
}
def test2(label: String) {
  assert(label == "persons", "Wrong Xml root element: " + xml)
  test3(label.length)
}
def test3(size: Int) {
  assert(size == 7, "Wrong Xml root element: " + xml)
}
```

Fig. 6. Matrix multiplication (for simplicity, we assume square matrices).

Bytecode inspection shows that all boxing has been removed, and closures have been inlined. The speedup can be seen in Figure 5.

5 Future work

The analysis used by the inliner is expensive, and we are looking at ways to speed it up. Since inlining is applied repeatedly, we think a form of incremental data flow analysis should help. Currently, at each step, after one method has been

inlined, a solution for the data flow equations is recomputed from scratch. The control flow graph is changed by replacing one method call by its own control flow graph, so large parts are similar. We believe a solution based on reusing partial results from previous solutions can speed it up considerably.

References

1. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. pages 324–341.
2. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.
3. Bill Venners Martin Oderksy, Lex Spoon. *Programming in Scala.* artima, 2008.
4. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, August 1997.
5. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
6. Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, StÃĭphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.
7. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.